PC-2024/25 Finalterm Assignment

D'Alò Gabriele
gabriele.dalo@edu.unifi.it

Abstract

This assignment is about the k-nearest neighbors algorithm carried out on datasets of various sizes. The main goal is to evaluate the difference in performances between sequential and parallel implementations by using a GPU.

1. Introduction

1.1. Assignment

In this assignment we want to use the KNN algorithm to answer queries. It is required to implement the algorithm sequentially and in parallel in order to evaluate the difference in the performances of the two implementations. We want to use the GPU to parallelize the sequential version. It is also required to carry out experiments on different cases, for example we want to test with datasets of different sizes and with different parallelization methods.

1.2. Objectives

The goal of this experiment is to become familiar with the use of the GPU and see how the performance of an algorithm implemented on the GPU varies.

1.3. Development environment

The operating system of the machine used is Windows 11 Home. The code has been written using the C++ language and Visual Studio with MSVC compiler was used as the IDE. To make use of the GPU the CUDA toolkit was used, below are some of the main features of the GPU used (the rest can be found in the NVIDIA documentation):

- GPU: NVIDIA GeForce RTX 4060 Laptop GPU
- Compute capability: 8.9 (Ada Lavelace)
- N° of SM: 24
- Max grid size: (2147483647, 65535, 65535)
- Max threads per block: 1024
- Warp size: 32
- Costant memory dimension: 64 KB
- Shared memory per block: 48 KB
- Shared memory per SM: 100 KB
- Total shared memory: 2400 KB
- Global memory: 8187 MB

2. Implementation

2.1. Structure

The project is made up of two main parts: one written in standard C++ which takes care of loading the data necessary for the experiments and executing the KNN in sequential version (.cpp files), and one written in C++ using CUDA which takes care of parallelising the execution of the code by bringing it to the GPU (.cu files).
Each dataset is made up of a certain number of vectors of a certain size (always 128 in our case) and comes with a certain number of query vectors (of the same size) to execute on it. The KNN algorithm consists in finding the K vectors of the dataset closest to each query vector (in our case K = 100), using the Euclidean squared distance as a metric. Some datasets are also

provided with groundtruth vectors to compare the results obtained.
Datasets are taken from Textmex ([1]) with sets: ANN_SIFT10K and ANN_SIFT1M, downloaded from `http://corpus-texmex.irisa.fr/`.

2.2. General overview

This project gives the possibility to carry out experiments on one of three datasets:

- **Small**: it is a dataset made up of 10000 data vectors and 100 query vectors (as mentioned, all vectors have 128 components). It is provided with groundtruth vectors to verify the accuracy of the results;
- **Big**: it is made up of 1000000 data vectors and 10000 query vectors, this also has groundtruth vectors;
- **Medium**: includes 300000 data vectors and 5000 query vectors. This dataset is taken as a subset of the Big dataset and therefore has no groundtruth vectors. The accuracy of the results is verified by comparing the results obtained from the sequential and parallel implementation.

Once the dataset has been selected it will be loaded, then the user has the possibility to select the GPU parallelization method to use (see 2.3 for details), after which the program will first execute the KNN in parallel version on the GPU, save the results obtained and the times required: the total one and separately the one to calculate all the distances from the queries and to sort the results in order to select only the closest K ones; and then it will run the sequential version to finally calculate the speedup obtained with the parallel version. Remember that:

$$Speedup = \frac{Execution\ time\ sequential}{Execution\ time\ parallel}$$

As mentioned, a check is also carried out on the results obtained to check that they are correct.
It should be noted that the total time of the parallel version also includes all the time needed for the various overheads such as launching the kernels, transferring data to/from the GPU, loading the data into the GPU memory and freeing it once finished. However, the time needed to check the accuracy of the results is not taken into account in the times.

2.3. Main project

Let's now move on to the main part of the project, namely the implementation of the parallel version on GPU. Ideally, when we execute code on the GPU we want to do it efficiently, organizing the blocks of the grid in order to exploit the greatest number of threads available in parallel, reducing overheads and synchronizations as much as possible. The reasons why the following implementations were chosen will be discussed in the next section (see 2.4).
First of all, we check the amount of free memory available on our GPU to save the results and if it is not enough to save them all, we divide the queries into batches, so as not to have to save the results all together. Depending on the number of queries it is also possible to save them in the constant memory of the GPU instead of in the global one so that they can be accessed more quickly.
As mentioned, the user has the possibility to choose between three techniques to use for parallelization, whose performances can vary significantly depending on the characteristics of the GPU available (for example the quantity of shared memory or the quantity of threads per block).
The **first version** organize the blocks by placing the components of a single vector along the x axis and the different vectors along the y axis, calculating how many vectors I can insert at most into a block, in order to maximize the number of threads that execute together without exceeding the amount of shared memory available. It organizes the blocks of the grid on the x axis considering the total size of the dataset and the number of vectors per block, while on the y axis it inserts the size of the query batch. This version makes use of **shared memory** to save partial distance results which it will eventually combine using parallel reduction to obtain the total distance for each vector in the block then saving it in global memory. The problem with this method is that the shared memory is limited and has a size, in general, quite small so it is not suitable for storing large quantities of data, even more so when we have a large number of grid blocks. This is

because, although each block has its own shared memory, the total available shared memory is limited and depends on the number of SMs. When we no longer have available memory, a block is forced to wait for it to be freed, therefore the execution time increases and the degree of parallelism is reduced.
The **second version** does not use shared memory but organizes the blocks of the grid in the same way as before, except that the number of vectors per block now no longer depends on the amount of shared memory available but only on the number of threads per block available (in our case we have a maximum of 1024 threads per block and 128 components for a vector, so we can put 8 vectors in a block). This version directly stores the results in global memory using an **atomic addition** operation to sum the partial results of each thread. The problem with this method is that atomic operations are problematic, especially on GPUs, as they block the execution of the threads in case someone is already accessing the same memory location requested, so they can lead to an increase in execution time due to the need for mutual exclusion to guarantee data consistency.
Finally, the **third version** does not use atomic operations or shared memory but ensures that each thread of a block calculates the distance between a vector of the dataset and a query vector and saves the final result directly in global memory. This reduces parallelism as now each thread has to carry out more operations since it does not work on the single component, however there is no longer anything blocking it until the end of its execution.
The methods described above are used to calculate the distance from each query for each vector in the dataset; once this is done it is necessary to select only the K vectors in the dataset closest to each query vector. To do this, we use the thrust library, specific for parallel frameworks such as CUDA, which provides a parallel sorting algorithm that allows you to sort the data according to a criterion that can be customized. Thrust has a fast-path radix sort which it will use in certain situations, in other cases, e.g. when you provide a custom functor, thrust will often use a slower merge-sort method.
The sequential implementation, instead, is quite standard, the distance between each element of the dataset and the query is calculated with an iterative cycle, after which the standard std sort is applied (which use quicksort, or usually a hybrid algorithm like introsort which combines quicksort, heapsort and insertion sort) and only the K vectors with the smallest distance are extracted, the process is repeated for each query sequentially.

2.4. Implementation choices

Let's now move on to describing why the implementation choices described above were made.
First of all, one thing that was not mentioned is that it was decided to use an AoS (Array of Structures) to save the neighbors, this was done because, in this case, when we go to save the neighbors or want to access them (such as when we do the sorting) we are always interested in obtaining both the index of the vector and its distance therefore having them saved as AoS favors access to nearby memory locations.
Now, the implementations reported in the final code are those that turned out to be the best from a performance point of view, but others were also tested. As regards the calculation of distances, the best implementation in theory would be the one that uses a thread for each component of each vector in order to maximize parallelism, however in this case it is necessary to save the partial distances calculated by each thread in order to then be able to calculate the total distance, but this represents a limit since for the reasons already mentioned it is not possible to save the partial results in shared memory and then only the total in the global memory. Therefore it is necessary to save the results directly in the global memory, but if each thread works on a component, when saving the result it will have to access the same memory area as the other threads working on the other components of the same vector and this leads to mutual exclusion problems. All this led to considering the third implementation which reduces parallelism as each thread works on more than one component, but also reduces the waiting times of the threads.
However, as regards the part of the sorting to extract the KNNs on the basis of the distances, the situation is more complex, since for one reason or another it is not possible to obtain the desired results in terms of speedup (see 3). In addition to the implementa-

tion present in the code, which saves all the distances between each vector and each query in global memory and then sorts them by extracting only the Ks at the shortest distance, other techniques have also been tested which however proved to be ineffective. In particular, I tried to:

- separate the results by distances and indices so you can use a native thrust comparator which in theory should be more efficient such as a sort_by_key method with a standard comparator, but this method turned out to be slower in our case;
- use radix sort by implementing it with CUB (a CUDA component). However, this method required more (temporary) memory and therefore it was necessary to process queries in even smaller batches if you had large amounts of data which results in more kernel launches, thus leading to an overall increase in the execution time;
- implement a heap-sort so you don't have to first sort all the elements and then extract only the best K ones. But even in this case the problem lay in where to save all the various min-heap structures and in the time required to then combine them all together;
- the various methods were also tested by transferring the data to the CPU where there is more space to save them and then transfer them back to the GPU. But, as expected, this causes an increase in time due to the overhead of transferring data between the CPU and GPU.

2.5. ANN retrival

For datasets with a lot of data, with high-dimensional vectors, it would actually be more convenient to perform an **approximate nearest neighbor** algorithm, i.e. an approximate method instead of an exact KNN search, thus reducing the time required. In particular, one method of ANN is to use locality sensitive hashing (LSH), which in short, consists of creating various hash tables divided into buckets and calculating its hash value for each point in the dataset by inserting it into the corresponding bucket. Once this is done, the hash value of the queries is also calculated and we see which bucket they would fall into, after which all the points saved in the same bucket are extracted as possible candidates and the true distance from the query is calculated. The general idea is that nearby points are mapped into the same bucket.
A search with LSH would, in theory, be more suitable for a GPU as it requires less memory and the calculations to be done are much fewer. However, I have not found a library available to implement LSH and make it parallelizable on GPU. The code contains the sequential version of ANN with LSH which uses the mlpack library (`https://www.mlpack.org/`), but this library seems to carry out all the various operations internally, making only a few APIs available to interact with the operations we would like to parallelize. This could be due to the fact that mlpack gives the availability to activate parallelization but using OpenMP which is a library for parallelization on CPU.

3. Resutls analysis

To conclude, let's analyze the results obtained from our experiments. As regards performance, the main parameter we are interested in analyzing is certainly the speedup which allows us to understand how quickly our parallel code performs compared to the sequential version. Our program has two main parts that we are interested in analyzing, the calculation of the various distances between data and queries and the sorting to extract the K best neighbors, therefore the two speedups are calculated separately in order to see which part is best parallelizable. We also analyze the total speedup on the execution of the entire algorithm since the parallel version introduces overhead for kernel launch, data transfer and synchronizations that the sequential version does not have.
Apart from this, using profiling tools such as Nsight compute provided by NVIDIA, it is also possible to observe other parameters such as the actual number of kernels launched and their execution times, grid and block sizes. Thanks to this tool, for example, it was possible to see that the version of thrust with radix-sort sorting launched many more kernels than the version with merge-sort and therefore required much more time than the implemented sorting version.

3.1. Results

The tables below show the speedup obtained compared to the sequential version on each of the three datasets for each parallelization technique used (with shared memory, with atomic addition and with one thread per vector). The accuracy obtained is also reported, remembering that for the small and big datasets this is calculated with respect to the groundtruth vectors supplied together with the dataset, while for the medium it is calculated by comparing the results obtained from the sequential and parallel versions.

3.1.1 Small Dataset

Method	Distances	Sorting	Total	Accuracy (%)
Shared	0,90	0,27	0,07	100
Atomic	1,29	0,27	0,07	100
Vector	9,36	0,27	0,07	100

Table 1. Parallelization speedup on small dataset

We immediately observe that the best version to parallelize the distance computation seems to be the third one, while the one that uses shared memory is the worst. This is due to the fact that even if the dataset is relatively small, the shared memory of the GPU used is not large enough to save the results for all blocks and therefore parallelism is reduced. Looking instead at the speedup on sorting we observe that it is the same in all cases, this is due to the fact that the operations and data on which it works are the same therefore it does not depend on the method used to calculate the distances.
As regards the overall speedup (which also includes the times to move the data between the memories, clean the GPU, etc...) we observe that for this dataset it is the same in all cases and is very low, the parallel version in this case is much worse than the sequential one. This is probably due to the fact that, for this dataset, "external" operations such as moving the data take much longer than those specific to the KNN (distance calculation and sorting), operations that the sequential version does not have to perform and is therefore much faster.

3.1.2 Medium Dataset

Method	Distances	Sorting	Total	Accuracy
Shared	1,43	2,46	1,81	100
Atomic	1,81	2,46	2,07	66,07
Vector	15,18	2,46	3,96	100

Table 2. Parallelization speedup on medium dataset

In this case the results are more or less the same but now we manage, even if slightly, to improve the sequential version. For the first and second methods we do not obtain a significant increase compared to the speedup on distance calculation, while for the third method we do, this may indicate that the third method scales much better than the others.
Another thing we can notice is that the precision of the version with atomic addition is not very high, this may be due to the fact that multiple atomic additions can influence the precision of the result.

3.1.3 Big Dataset

Method	Distances	Sorting	Total	Accuracy
Shared	1,45	3,07	2,04	99,98
Atomic	1,85	3,07	2,36	51,57
Vector	14,86	3,07	4,63	99,98

Table 3. Parallelization speedup on big dataset

Even with the largest dataset, the situation is the same as in the other cases, the definitively best version therefore seems to be the one that calculates the distance between an entire vector and a query using a thread instead of one for each component of the vector.
We also observe that the precision of atomic is even lower, this is because the dataset is larger and therefore more operations are performed. Even the accuracy of the other methods is not exactly 100% but this is probably due to the sorting criterion used in case of vectors with equal distance. But the most important thing we can notice is that the speedup of the best version is not improved compared to the previous case, this may indicate that this implementation also has a limitation on its scalability.

3.2. Performances evaluation

Making a final evaluation on the results we can conclude that the best version for our problem is the

one that uses one thread per vector, that sacrifices a bit of parallelism to reduce the idle times of the threads that have to work. Despite this, the overall speedup is quite low, this is due to the fact that sorting represents a bottleneck in this case as it was not possible to find a more effective sorting algorithm due to memory limitations.

4. References

References

[1] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.